Linked Data (in resourceless) Platforms: a mapping for Constrained Application Protocol

Giuseppe Loseto, Saverio Ieva, Filippo Gramegna, Michele Ruta*, Floriano Scioscia, and Eugenio Di Sciascio

Politecnico di Bari - via E. Orabona 4, Bari (I-70125), Italy
{giuseppe.loseto,saverio.ieva,filippo.gramegna,michele.ruta,
floriano.scioscia,eugenio.disciascio}@poliba.it

Abstract. This paper proposes a mapping of the Linked Data Platform (LDP) specification for Constrained Application Protocol (CoAP). Main motivation stems from the fact that LDP W3C Recommendation presents resource management primitives for HTTP only. Hence, use cases related to Web of Things scenarios, where HTTP-based communication and infrastructures are unfeasible, are partially neglected. A general translation of LDP-HTTP requests and responses is provided, as well as a fully comprehensive framework for HTTP-to-CoAP proxying. The theoretical work is corroborated by an experimental campaign using the W3C Test Suite for LDP.

Keywords: Linked Data Platform, CoAP, Semantic Web of Things

1 Introduction and Motivation

The World Wide Web Consortium (W3C) has standardized the Linked Data (LD) management on the Web: the Linked Data Platform (LDP) specification [8] gives guidelines for classifying resources according to their type. The final aim was to improve previous RDF graphs management based on SPARQL 1.1 Graph Store HTTP protocol[1] also fixing multiple issues. Unfortunately, this effort leaves out the so-called Web of Things (WoT) where HTTP is replaced by more simple application protocols as for example CoAP (Constrained Application Protocol) [11], a layer 7 standard suitable for Machine-to-Machine (M2M) communication in resourceless scenarios. CoAP adopts a loosely coupled client/server model, based on stateless operations on *resource* representations [2]. Each resource is unambiguously identified by a URI (Uniform Resource Identifier) and clients access them via asynchronous request/response interactions based on HTTP-derived methods mapping the Read, Create, Update and Delete operations of data management.

Sec. 3.12 of Linked Data Platform Use Cases and Requirements [1]) reports on a possible one-to-one translation of HTTP primitives toward CoAP, nevertheless the proposed solution appears quite limited. The given mapping [4] only

* Corresponding author. Tel.: +39-339-635-4949; fax: +39-080-596-3410
[1] https://www.w3.org/TR/sparql11-http-rdf-update/

considers basic HTTP interactions: several methods and headers are not used and/or too simplified. As an example the methods *options*, *head* and *patch* are not allowed as well as several MIME types (content-format) are missing. Hence, not negligible operations on resources introduced by LDP cannot be applied with the loss of significant LDP functionality.

Main motivation of this paper stems from the need of extending and enriching capillary the standardization of Linked Data Platforms to Web of Things use cases. We propose a specific variant of the HTTP-CoAP mapping able to preserve all the LDP features and capabilities with a full support of the W3C specification: the envisioned HTTP-CoAP proxy enables networks of objects to be included in the Web as first-class Linked Data providers. Novel features are also added giving value to the strongest peculiarities of CoAP (*e.g.*, resource discovery based on CoRE Link Format), with respect to HTTP.

The proposed solution is released as open source. Performance tests evidence LDP-CoAP supports all types of LDP resources keeping computational performances comparable with other frameworks, except for Non-RDF Source tests. Results of the W3C LDP conformance test suite show the proposal does not completely cover LDP specification yet.

The remainder of the paper is organized as follows. The next section frames the context the proposal refers to, while the following Section 3 presents the proposed LDP-CoAP mapping. Then Section 4 introduces the validation framework before Section 5 which addresses experimental campaign made to corroborate the approach. Finally, Section 6 closes the paper sketching future work.

2 Coping with Lightweight Linked Data Platform

2.1 Scenarios

The Internet of Things (IoT) and Web of Things (WoT) paradigms envision networking heterogeneous resource-constrained devices in order to enable information exchange in a wide range of pervasive computing scenarios. Supply chain management benefits from sensor-equipped RFID (Radio Frequency IDentification) tags: in addition to basic product tracking, they allow monitoring physical properties to prevent tampering and spoilage. Ubiquitous healthcare solutions exploit wearable medical devices for continuous health monitoring, with the ability to increase home care and automatically alert caregivers when issues occur, so reducing hospitalization time and related costs as well as improving patients' quality of life. *Smart city* solutions are another huge opportunity for pervasive information gathering, exchange and processing. They concern all aspects of the urban infrastructure, including the electrical power grid, road and transportation networks, environmental monitoring and more. IoT is also increasingly penetrating the industrial sector, where state-of-the-art factory control and automation infrastructures are increasingly based on IP (Internet Protocol) and require the coordination of large numbers of (wireless) sensor and actuator devices. This supports dynamic, flexible planning and scheduling approaches for industrial plant management.

All the above scenarios require managing data streams transmission and resource management, where *resources* include data sources, observed phenomenons and the devices themselves. Many different frameworks are available for managing devices and developing IoT applications. Unfortunately, they are often designed with limited cross-compatibility and federation capabilities. IoT/WoT development is therefore increasingly facing the same issues as Web application *mashups*, where most of the integration work must be done by hand on a case-by-case basis. Systematic, standardized and efficient information access and integration interfaces are increasingly needed as the number and variety of devices as well as the amount of produced information grow.

2.2 Linked data Platform on Constrained Application Protocol

The LDP W3C Recommendation provides standard rules for accessing and managing Linked Data on Web *LDP servers*. Basically, it defines seven types of LDP Resources as well as patterns of HTTP methods and headers for CRUD (Create, Read, Update, Delete) operations.
– **LDP Resource (LDPR)**: any HTTP resource complying with the basic LDP guidelines;
– **LDP RDF Source (LDP-RS)**: a LDPR which corresponds to an RDF graph and can be fully represented in an RDF syntax. LDP explicitly supports `text/turtle` [3] and `application/ld+json`[7] serializations;
– **LDP Non-RDF Source (LDP-NR)**: a LDPR not represented in RDF, *i.e.*, a binary or text document without RDF annotations. LDP servers can generate metadata about LDP-NR resources, *e.g.*, creation date or owner;
– **LDP Container (LDPC)**: a particular type of LDP-RS for grouping LDP resources. Three types of LDPCs are defined, namely *Basic*, *Direct* and *Indirect*;
– **LDP Basic Container (LDP-BC)**: an LDPC defining a simple link to its resources through the `ldp:contains` predicate;
– **LDP Direct Container (LDP-DC)**: a more flexible LDPC, containing *membership* triples: they specify the *membership resource* and the *member relation*;
– **LDP Indirect Container (LDP-IC)**: an LDPC similar to LDP-DC, but also capable of having member resources with unrelated URIs w.r.t. the container.

W3C LDP implementations web page[2] lists several software tools implementing the LDP Recommendation. Table 1 reports the most relevant ones, in order of release date. Main properties and supported resource types are summarized, as obtained from the *LDP implementation conformance report* [12]. All solutions are based on the HTTP protocol, with no current support for WoT standards such as CoAP.

The W3C suggests explicit use cases [1] aiming to integrate Linked Data Platform in scenarios related to resource-constrained devices and networks with specific reference to the Constrained Application Protocol (CoAP). CoAP was

[2] `http://www.w3.org/wiki/LDP_Implementations`

Table 1: Current LDP implementations

Name	**Status**	**Last Version**	**License**	**Language**	**Supported LDPR**
RWW.IO	Pending	1.2 (Nov 2014)	MIT	PHP	RS, BC
Apache Marmotta	Full release	3.3.0 (Dec 2014)	APL 2.0	Java	RS, NR, BC
Bygle	In progress	Feb 2015	APL 2.0	Java	RS, BC
Eclipse Lyo	Completed	2.1.0 (Mar 2015)	EPL 1.0	Java	RS, NR, BC, DC
LDP.js	Completed	Apr 2015	APL 2.0	JavaScript	RS, BC, DC
Glutton	In progress	Apr 2015	GPLv3	Python	RS, BC
Carbon LDP	In progress	0.5.7 (Oct 2015)	BSD	JavaScript	RS, NR, BC, DC, IC
LDP4j	In progress	0.2.0 (Dec 2015)	APL 2.0	Java	RS, BC, DC, IC
RWW Play	In progress	2.3.6 (Dec 2015)	APL 2.0	Scala	RS, NR, BC
Fedora	Full release	4.5.0 (Jan 2016)	APL 2.0	Java	RS, NR, BC, DC, IC
Callimachus	Full release	1.5.0 (Mar 2016)	APL 2.0	Java	RS, NR, IC
gold	In progress	1.0.1 (Apr 2016)	MIT	Go	RS, BC
OpenLink Virtuoso	Full release	7.2.5 (Apr 2016)	GPLv2	C/C++	RS, BC
ldnode	In progress	0.2.31 (Apr 2016)	MIT	JavaScript	RS, BC

conceived for machine-to-machine (M2M) communication in resourceless environments. Also in this case, the reader is referred to specifications [11] for protocol details. The CoRE Link Format specification [10] is adopted for resource discovery through the reserved `/.well-known/core` URI. CoAP also supports proxying, enabling Web applications to transparently access the resources hosted in devices based on CoAP. Some CoAP options are derived from HTTP header fields (*e.g.*, content type, headers for conditional requests and proxy support), while some other ones have no counterpart in HTTP. In summary, CoAP is basically a lightweight version of HTTP including some of its most important features into a more compact protocol, so an HTTP-CoAP mapping is needed to exploit all LDP features with CoAP. An early mapping proposal was defined in [4] but it only worked with basic HTTP interactions.

The novel HTTP-CoAP mapping for LDP developed here overcomes these shortcomings, also enabling a direct CoAP-to-CoAP interaction among devices supporting LDP-CoAP. *HTTP methods* mapping is applied for each CoAP method (if present). `PATCH`, `HEAD` and `OPTIONS`, undefined in CoAP, are mapped to existing `GET` and `PUT` methods, by adding the new Core Link Format attribute `ldp`. This solution ensures full backwards compatibility with the standard protocol, while extending the basic CoAP functionalities. *HTTP headers* of request/response messages are translated as in Table 2, also including LDP *prefer* headers. Additional *content-format* media types are introduced (`text/turtle` [3], `application/ld+json` [7] and `application/rdf-patch` [9]) to support LDP features.

Table 2: HTTP-CoAP mapping of headers

HTTP Header	**LDP-CoAP**
`Content-Type`	Content-Format (`ct`) CoAP option
`Link` (`rel="type"`)	Resource-Type (`rt`) Core Link Format attribute, available through a CoAP discovery request

`Allow`
`Accept-Post`
`Accept-Patch`
	Undefined in CoAP, available in JSON format as body content of an LDP-CoAP Options request

`Slug`	`title` Core Link Format attribute
`Location`	`location-path` CoAP option
`Prefer: return=representation; include="`*`pref`*`"`	`ldp-incl=`*`pref`* Core Link Format attribute
`Prefer:return=representation; omit="`*`pref`*`"`	`ldp-omit=`*`pref`* Core Link Format attribute
`Preference-Applied: return=representation`	*`pref`* returned using `location-query` CoAP option

3 LDP-CoAP mapping implementation

LDP-CoAP mapping was implemented in a Java-based framework providing the basic components required to build read-write applications and publish Linked Data on the WoT according to LDP-CoAP specification. The framework consists of several modules, as shown in Figure 1:

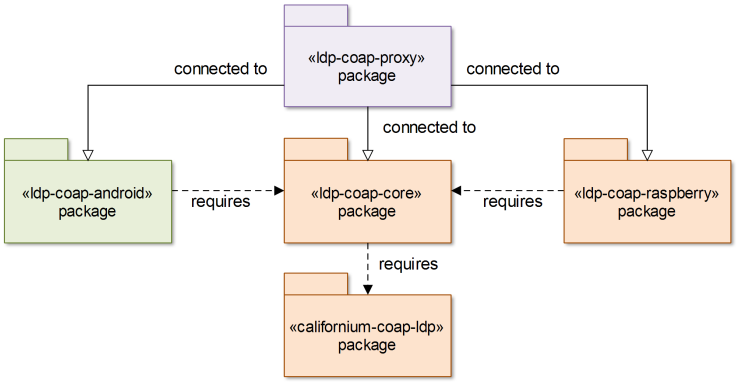

Fig. 1: Modules of LDP-CoAP framework

ldp-coap-core: includes the implementation of all LDP-CoAP resources and a basic LDP-CoAP server handling CoAP-based communication and RDF data management. The main Java package `coap.ldp` was partitioned in the following sub-packages to separate developed classes in well-defined sections each providing a specific functionality.
– `coap.ldp.server`: contains the reference `CoAPLDPServer` implementation. It extends the `CoAPServer` provided by *californium-core-ldp* module (described

below) and exposes several methods to create and manage LDP resources. The package also includes the `CoAPLDPTestSuiteServer`, used for experiments described in Section 5, and the `CoAPLDPServerMessageDeliverer` needed to implement the *PUT-to-create* method of LDP.
– `coap.ldp.resources`: according to the LDP resource hierarchy defined in [8], several Java classes were developed extending the `CoAPLDPResource` base class providing common methods and attributes. For each resource class, a specific data handler can be implemented to retrieve whatever kind of data (*e.g.*, observation from a sensor) and update the RDF repository with a user-defined sampling period. Handlers can be defined starting from the `LDPDataHandler` abstract class. In this way, developers can build specific applications implementing the whole business logic and data management procedures within the `handleData` method of the handler, without any other modification of the source code. `CoAPLDPResourceManager` implements read-write operations on the RDF data storage exploiting *OpenRDF Sesame*[3] for creating, querying and storing RDF triples using a local in-memory repository.
– `coap.ldp.handler`: two simple handlers were defined as usage examples to expose real-time system CPU load and RAM usage ratio as LDPRDFResource. Data are collected by means of management interfaces for the operating system provided by Java 7 (or above) JDK.
– `coap.ldp.exception`: a `CoAPLDPException` class was defined to catch specific errors related to an incorrect usage of LDP methods, headers or attributes. Several subclasses represent the most typical problems (*e.g.*, *content format* or *precondition failed*).
– `rdf.vocabulary`: an additional package containing RDF ontology files mapped as Java classes. Constants related to concepts and object properties for a given namespace can considerably simplify creation and querying of RDF triple. As an example, *SSN-XG* ontology [5] was mapped through the *Sesame Vocabulary Builder*[4] tool and included in the package.
The following libraries are also required to correctly compile the *ldp-coap-core* module:

- *JSON-java*[5] to format data in JSON;
- *jsonld-java*[6] to format data according to the `json-ld` specification [7];
- *Apache Marmotta RDF Patch Util*[7] to update RDF statements of a Sesame repository according to the `rdf-patch` [9] format;

californium-core-ldp: a modified version of the *Californium* CoAP framework [6], extended to support LDP features. Main modifications include: (i) novel content-format media types added to `MediaTypeRegistry` class; (ii) additional

[3] `http://rdf4j.org`
[4] `http://github.com/tkurz/sesame-vocab-builder`
[5] `http://github.com/stleary/JSON-java`
[6] `http://github.com/jsonld-java`
[7] `http://marmotta.apache.org/sesame.html`

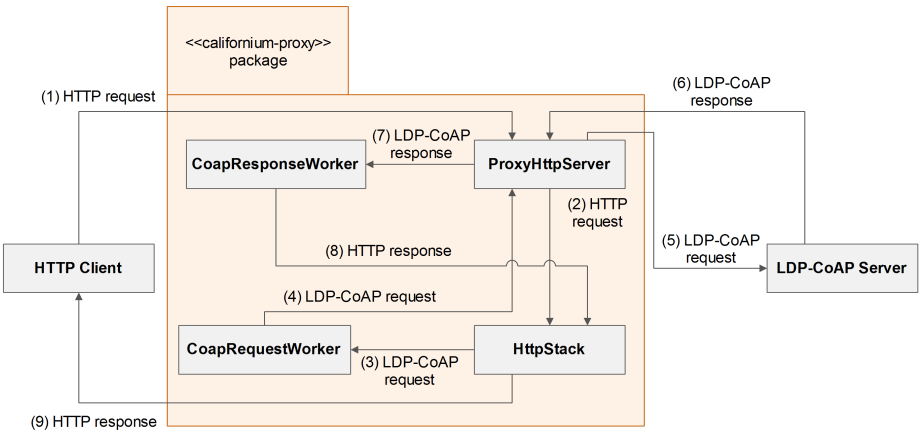

Fig. 2: LDP HTTP-CoAP Proxy Server

response codes introduced within `CoAP` main class; (iii) private attributes of `ServerMessageDeliverer` changed to protected to allow usage in subclasses.

ldp-coap-proxy: a modified version of *californium-proxy* implementing the mapping rules defined in Section 2.2. It is used to translate LDP-HTTP request into the corresponding LDP-CoAP ones. Based on LDP specification, LDP servers need to implement all mandatory methods. Methods unavailable in CoAP could be added from scratch to the CoAP protocol itself, however the proposed solution aims to enhance standard CoAP methods maintaining at the same time a full compatibility with the core protocol. As shown in Figure 2, LDP-CoAP mapping procedures take advantage of the classes in this module. In particular, *ProxyHttpServer* is responsible for processing a request –coming from a generic HTTP client– through its *HttpStack* member class where the LDP-CoAP mapping occurs. *HttpStack* transforms an HTTP request into a compatible LDP-CoAP one and for each CoAP request it starts two threads, *CoapRequestWorker* and *CoapResponseWorker*, synchronized according to the producer-consumer pattern. The *CoapRequestWorker* thread produces the LDP-CoAP translated request for the *ProxyHttpServer* class instance which forwards that request to the proper LDP-CoAP server. The *CoapResponseWorker* is responsible for consuming and translating the LDP-CoAP response coming from the *ProxyHttpServer* into the HTTP response which is returned to the client.

In addition to the basic framework, the following two packages were developed to build WoT applications based on LDP-CoAP on particular embedded and resource-constrained platforms.

ldp-coap-raspberry: *ldp-coap-core* was tested on a Raspberry Pi[8] board to prove the usability of the proposed framework also on embedded platforms. In comparison to other LDP implementations, LDP-CoAP is very lightweight and

[8] `http://www.raspberrypi.org`

simple to run on resourceless environments like Raspberry Pi, having a minimum number of dependencies and low system requirements in terms of memory and processing resources. As a reference example, two handlers were implemented to publish CPU temperature and free RAM as LDP resources. Data are retrieved using the *Pi4J*[9] library.

ldp-coap-android: a simple project exploiting *ldp-coap-core* on Android devices. It runs unmodified on all platforms supporting modules compiled with Java SE runtime environment, version 7 or later, so it can be directly used as a library also by Android applications. Android OS provides a uniform interface[10] to access sensor data. Therefore, a single sensor handler (named `GenericSensorHandler`) was implemented to manage both hardware and software-based device sensors by specifying just the type of sensor to query. The project includes a basic activity starting a LDP-CoAP server exposing data from environment sensors (light and pressure), motion sensors (accelerometer, gyroscope and step counter) and position sensors (proximity and orientation) modeled as LDPR, LDP-BC or LDP-DC. *ldp-coap-android* was developed using Android SDK Tools (Revision 25.1.1) with reference API level 22, thus it is compatible with all devices running Android 5.1.1 or later.

Source code for the whole framework is available on the *GitHub* repository[11]. All modules were developed as Eclipse[12] projects using Apache Maven[13] to manage Java dependencies and required libraries. Only *ldp-coap-android* is a project for Android Studio[14], the Google official IDE for app development. In this case, all dependencies can be defined through a Gradle[15] configuration file.

4 Validation examples

Validation of the proposed approach occurred by showing that the LDP-CoAP mapping can cover functionalities of LDP over HTTP. Full examples for each HTTP method are on the LDP-CoAP project Web page[16]. Due to space constraints, a few reference examples are discussed here, in order to clarify the proposal. Note that in some cases an HTTP request cannot be translated into a single CoAP request, but more CoAP messages are needed.

Example 1. Basic HTTP GET request on an LDP resource

[9] `http://pi4j.com`

[10] Android sensor framework, `http://developer.android.com/guide/topics/sensors/sensors_overview.html`

[11] Repository URL: `http://github.com/sisinflab-swot/ldp-coap-framework`, persistent URL of v1.0 release: `http://dx.doi.org/10.5281/zenodo.50701`

[12] `http://eclipse.org`

[13] `http://maven.apache.org`

[14] `http://developer.android.com/tools/studio/index.html`

[15] `http://gradle.org`

[16] `http://sisinflab.poliba.it/swottools/ldp-coap`

```
GET /alice/ HTTP/1.1
Host: example.org
Accept: text/turtle
```

The HTTP response is shown in Figure 3. In this case, a single CoAP `GET` request is not able to produce all the required headers, because some of them are not defined in the response format of CoAP. So the original HTTP request is translated to the following three LDP-CoAP requests:
- a `GET` message to map Content-Format (`ct`), `ETag` (if present) and RDF content of the LDP resource;
- a CoAP discovery message to retrieve the `rt` attribute indicating the LDP type of each resource. It maps the HTTP `Link` response header;
- an `OPTIONS` message (described later) to map the `Allow`, `Accept-Post` and `Accept-Patch` response headers.

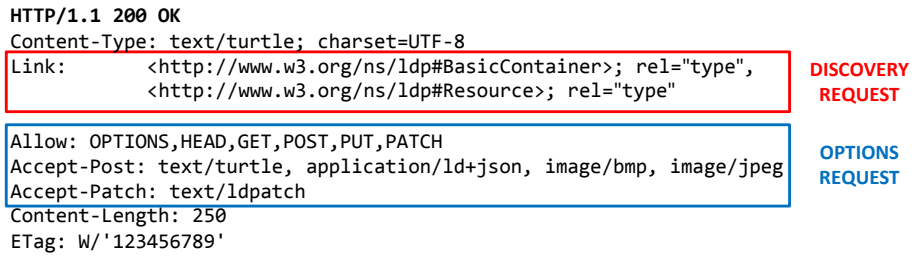

Fig. 3: Example 1 – HTTP GET response (payload data not included)

Example 2. Create a new LDP resource through an HTTP POST request

```
POST /alice/ HTTP/1.1
Host: example.org Slug: foaf
Content-Type: text/turtle <payload>
```

In this case, the request is translated to a single CoAP POST message with URL:

```
coap://example.org/alice?title=foaf&rt=ldp:Resource
ct=text/turtle <payload>
```

`title` and `rt` query parameters are obtained from the `Slug` and `Link` HTTP header fields, respectively. If the `Link` header is not defined, `ldp:Resource` is used as default value of `rt`. Finally, as shown in Figure 4, the HTTP POST response will contain the `Location` and `Link` headers corresponding, to the `Location-Path` and `Location-Query` CoAP response options.

Example 3. HTTP OPTIONS request on an LDP resource

An `OPTIONS` request is used to obtain useful information about a resource, *e.g.*, the list of applicable methods. An example HTTP `OPTIONS` response is shown in Figure 5. Also in this case, multiple LDP-CoAP requests are combined to produce the HTTP reply with all required headers:
- `Allow`, `Accept-Post` and `Accept-Patch` response headers are not defined in CoAP, so their values are set in the LDP-CoAP `OPTIONS` response body in JSON syntax and

```
HTTP/1.1 201 Created
Location: http://example.org/alice/foaf
Link: <http://www.w3.org/ns/ldp#Resource>; rel="type"
Content-Length: 0
```

Fig. 4: Example 2 – HTTP POST response

then mapped to the corresponding HTTP headers;
- a CoAP discovery request is used to obtain the resource type (`rt`) then mapped to the HTTP `Link` response header.

```
HTTP/1.1 204 No Content
Allow: OPTIONS,HEAD,GET,POST,PUT,PATCH
Accept-Post: text/turtle, application/ld+json, image/bmp, image/jpeg
Accept-Patch: text/ldpatch
Link: <http://www.w3.org/ns/ldp#BasicContainer>; rel="type",
<http://www.w3.org/ns/ldp#Resource>; rel="type"
```

DISCOVERY REQUEST

Fig. 5: Example 3. HTTP OPTIONS response

5 Experiments

The W3C LDP Test Suite[17] was used to evaluate the functionality of the proposed framework and to compare it with existing solutions. The suite directly queries an LDP server by means of HTTP messages; only for LDP-CoAP tests requests were sent to the server through an LDP-CoAP proxy as in Figure 1. The suite consists of 236 tests referred to rules and restrictions of the LDP W3C specification. Obtained results are grouped by supported LDP resources: RDF Sources, Non-RDF Sources and Basic, Direct, Indirect Containers (see [8] for definitions). For each test category, the specification requirements are divided in three compliance levels: MUST, SHOULD, and MAY. As reported in Table 3, LDP-CoAP results were compared with other LDP implementations. All those tools are based on HTTP only. For each resource/level pair, the score of LDP-CoAP is shown along with the highest value obtained by other tools. Overall, LDP-CoAP presents good scores, when considering 17 manual tests were skipped in this first experimental campaign and only automated ones were executed.

Performance evaluation was carried out exploiting requests of the test suite to query 7 tools in addition to LDP-CoAP: *Virtuoso*, *LDP.js*, *Apache Marmotta*, *LDP4j*, *RWW.IO*, *Fedora4* and *Eclipse Lyo*. They were selected according to the features listed in Table 1: current status, completeness, open license, last update and supported resources (in particular RDF Source and Basic Container). Also *gold* was tested but then discarded due to the limited compatibility with LDP specification. For each software, only supported resources were tested to retrieve the overall request processing time. Each test was repeated three times on the same PC and (only for tests passed by all tools) the average value was taken and grouped by resource type. Results are reported

[17] `http://w3c.github.io/ldp-testsuite/`

Table 3: Comparison of implementation conformance tests

Feature	**LDP-CoAP**	**Highest Score**
RS – must	91.7% (22/24)	100% (Callimachus, Eclipse Lyo, Apache Marmotta, LDP4j, LDP.js)
RS – should	71.4% (5/7)	100% (Callimachus, Eclipse Lyo, Apache Marmotta, LDP4j)
RS – may	100% (1/1)	100% (Callimachus, Eclipse Lyo, Fedora4, Apache Marmotta, ldphp, LDP4j, Virtuoso, rww-play, LDP.js)
BC – must	86.5% (32/37)	100% (Eclipse Lyo, Apache Marmotta, LDP4j, LDP.js)
BC – should	88.2% (15/17)	100% (Eclipse Lyo, Apache Marmotta)
BC – may	100% (4/4)	100% (Eclipse Lyo, Fedora4, Apache Marmotta, LDP.js)
DC – must	88.1% (37/42)	100% (Eclipse Lyo, LDP4j, LDP.js)
DC – should	89.5% (17/19)	100% (Eclipse Lyo)
DC – may	100% (4/4)	100% (Eclipse Lyo, Fedora4, LDP4j)
IC – must	84.6% (33/39)	97.4% (Callimachus)
IC – should	88.2% (15/17)	88.2% (LDP4j)
IC – may	100% (4/4)	100% (Fedora4)
NR – must	80% (12/15)	100% (Callimachus, Eclipse Lyo, Apache Marmotta)
NR – should	100% (1/1)	100% (Callimachus, Eclipse Lyo, Fedora4, Apache Marmotta, rww-play)
NR – may	66.6% (4/6)	100% (Eclipse Lyo, Fedora4, Apache Marmotta)

in Figure 6. Fedora4 and LDP-CoAP support all types of LDP resources, so resulting very competitive. Eclipse Lyo and LDP4j are able to manage four groups of resources, whereas the other frameworks basically perform only operations on RDF Sources and Basic Containers. LDP-CoAP exhibits good processing times, as results are comparable with the other implementations even while involving the HTTP-CoAP proxy. Only for Non-RDF Source tests, performance is slightly worse.

In order to evaluate the feasibility of exploiting LDP in mobile and pervasive computing scenarios, LDP-CoAP performance was tested on three different reference platforms: a PC testbed[18], an Android smartphone[19] and a Raspberry Pi 1 Model B+ board[20]. All requests were originated from a PC client running both the LDP Test Suite and the LDP HTTP-CoAP proxy, connected through a local IEEE 802.11 network to one of the three LDP-CoAP servers for each test. The overall processing time, reported in the following tables, is defined as the time elapsed from sending the request until receiving a response by the client, including communication and HTTP-CoAP message translation times. Figure 7 shows processing times on PC, Android and Raspberry. Values on Android are roughly 3 times higher than on PC, whereas performance on Raspberry are an order of magnitude higher with respect to PC. However, average response times are under 1 second both on Android and Raspberry (except for

[18] Equipped with Intel Core i7 CPU 3770K at 3.50 GHz (4 cores/8 threads), 12 GB DDR3-SDRAM (1333 MHz) memory, 2 TB SATA (7200 RPM) hard disk, 64-bit Microsoft Windows 7 Professional and 64-bit Java 8 SE Runtime Environment (build 1.8.0_65-b17).

[19] LG Google E960 *Nexus 4* with quad-core Qualcomm APQ8064 Snapdragon S4 Pro CPU at 1.5 GHz, 2 GB RAM, 16 GB internal storage memory and Android 5.1.1.

[20] Equipped with a single-core ARM11 CPU at 700 MHz, 512 MB (shared with GPU) RAM, 32 GB SD card, Raspbian Wheezy OS and 32-bit Java 8 SE Runtime Environment (build 1.8.0-b132)

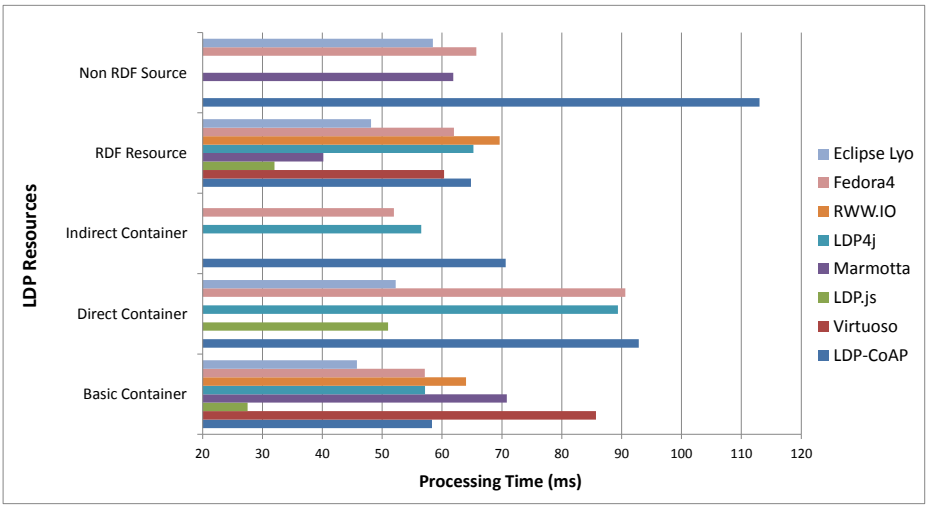

Fig. 6: Comparison of processing time for tested LDP implementations

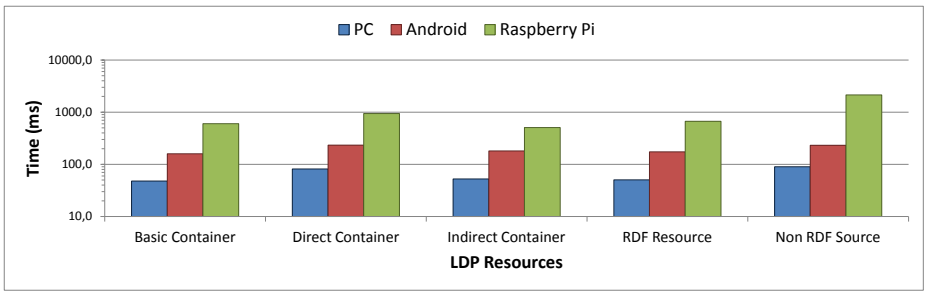

Fig. 7: Comparison of LDP-CoAP processing time on different platforms

LDP-NR responses on Raspberry) so they can be deemed as acceptable in mobile and resourceless contexts.

Memory usage was also measured every 2 s during the execution of the test suite for the three platforms. The memory allocation peak of the LDP-CoAP server was about 44.7 MB on PC, 18.3 MB on Android and 7.4 MB on Raspberry. This is due to the stricter memory constraints on smartphones and embedded devices, imposing to have as much free memory as possible at any time. Consequently, on these platforms Java virtual machines perform more frequent and aggressive garbage collection, as reported in Figure 8 showing memory usage on Raspberry Pi. It is possible to notice the garbage collector was invoked many times, corresponding to the falling edges in the chart. This behavior reduces memory usage, but on the other hand it can be responsible for a significant portion of the processing time gap found on the different platforms.

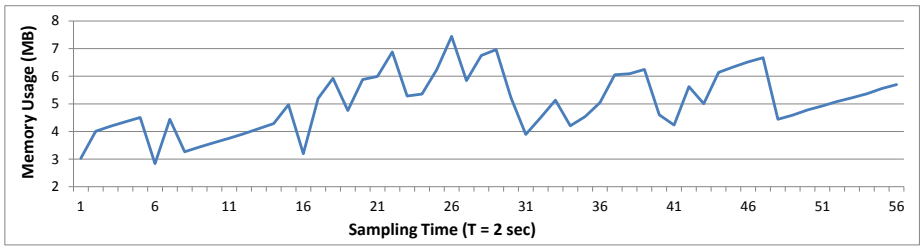

Fig. 8: Memory Usage of LDP-CoAP on Raspberry Pi

6 Future directions

This work presented an LDP-CoAP mapping and framework for managing Linked Data in the Web of Things. It allows practical extension of the LDP paradigm to resource-constrained platforms. The proposal includes an HTTP-CoAP proxy to allow networked objects to be included in the Web as first-class Linked Data providers. The working solution, publicly released as open source, allows demonstrating and evaluating the approach. Performance tests evidence LDP-CoAP supports all types of LDP resources and its computational performances are comparable with those of other frameworks, except for Non-RDF Source tests. Results of the W3C LDP conformance test suite show the proposal does not cover every requirement of the LDP specifications yet.

Future revisions will extend compliance as much as possible; progress will be measured through the same test suite. Planned developments also include: evolving the forks of Californium core and proxy modules, in order to merge them with the original codebase eventually; adding the capability to manage RDF resources on persistent storage in addition to in-memory ones; studying optimized solutions for RDF compression of small-size resources. Porting the LDP-CoAP server to more computing platforms (*e.g.*, Arduino) for smart objects is a further goal, in order to enable developers to create WoT solutions in heterogeneous scenarios. Finally, both case studies and user feedback will allow assessing the practical effectiveness of LDP in pervasive computing and improving the proposed implementations.

References

1. Battle, S., Speicher, S.: Linked Data Platform Use Cases and Requirements. W3C Working Group Note, W3C (Mar 2014), http://www.w3.org/TR/ldp-ucr/
2. Bormann, C., Castellani, A., Shelby, Z.: CoAP: An Application Protocol for Billions of Tiny Internet Nodes. IEEE Internet Computing 16(2), 62–67 (2012)
3. Carothers, G., Prud'hommeaux, E.: RDF 1.1 Turtle (Terse RDF Triple Language). W3C Recommendation, W3C (Feb 2014), http://www.w3.org/TR/turtle/
4. Castellani, A., Loreto, S., Rahman, A., Fossati, T., Dijk, E.: Guidelines for HTTP-CoAP Mapping Implementations. Internet-Draft draft-ietf-core-http-mapping-07, IETF Secretariat (July 2015)
5. Compton, M., Barnaghi, P., Bermudez, L., Garcia-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A., et al.: The SSN Ontology

of the W3C Semantic Sensor Network Incubator Group. Web Semantics: Science, Services and Agents on the World Wide Web 17 (2012)
6. Kovatsch, M., Lanter, M., Shelby, Z.: Californium: Scalable cloud services for the internet of things with coap. In: Internet of Things (IOT), 2014 International Conference on the. pp. 1–6. IEEE (2014)
7. Lanthaler, M., Sporny, M., Kellogg, G.: JSON-LD 1.0 (A JSON-based Serialization for Linked Data). W3C Recommendation, W3C (Jan 2014), http://www.w3.org/TR/json-ld/
8. Malhotra, A., Arwe, J., Speicher, S.: Linked Data Platform 1.0. W3C Recommendation, W3C (Feb 2015), http://www.w3.org/TR/ldp/
9. Seaborne, A., Vesse, R.: RDF Patch Describing Changes to an RDF Dataset. Unofficial Draft (Aug 2014), https://afs.github.io/rdf-patch/
10. Shelby, Z.: Constrained RESTful Environments (CoRE) Link Format. RFC 6690 (Proposed Standard) (Aug 2012), `http://www.ietf.org/rfc/rfc6690.txt`
11. Shelby, Z., Hartke, K., Bormann, C.: The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard) (Jun 2014), `http://www.ietf.org/rfc/rfc7252.txt`
12. Speicher, S., Fernández, S.: Linked Data Platform Implementation Conformance Report. W3C Working Group Note, W3C (Dec 2014), http://www.w3.org/TR/ldp-implreport/